TÉCNICO LISBOA

Robotics Report Lab 2

Claudio Correia (81959)
Marta Freitas (82017)
Martijn Copier (91330)

Instituto Superior Técnico
Lisbon, Portugal

June 3, 2018

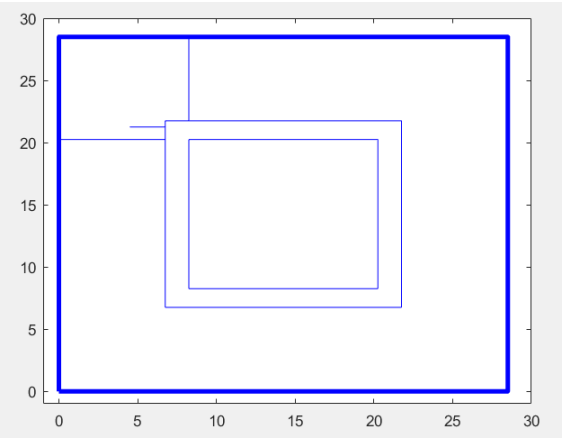
Introduction

Our assignment was to control a Pioneer DT robot to navigate around the North towers' 5th floor, starting from inside the lab and returning to the starting point.
The solution developed to accomplish this task is based on following a trajectory from the environment, which is known beforehand.

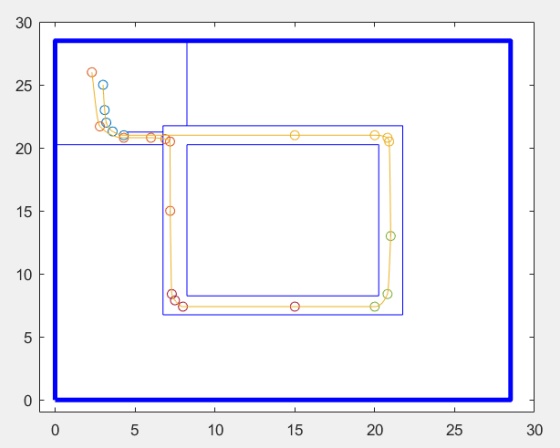
Models and Algorithms

Model of the 5th floor

Using the measurements from the floor plan, we created a model. This model was used to approximate the real world layout of the floor. The numbers on the axes represent meters.



Floorplan without points



Floorplan with Points

Calculating the Trajectory

From the floor plan of the 5th floor, we chose different points and translated them into our programs' world coordinates (as can be seen in the right picture above). After this we used the function *pchip*, this function interpolates all the points and creates a trajectory that the robot will follow.
The chosen points were iteratively updated during the development of the project, just as the offsets that accompanied them.

Calculating the Coordinates

There are two coordinates systems to consider, one from the point of view from the robot and one which represent the global variables from our world. The coordinates are (0,0) and (3, 23.5), respectively. Each time the robot moves we have to translate it's relative coordinates com the (0,0) of where it thinks it started to it's actual position in the world. This transformation also rotates the coordinate system by $\frac{\pi}{2}$.

Following the Trajectory

When the robot is initialized, the Euclidean distance to the 150 next points of the trajectory is calculated and we chose the closest one. This point becomes the new objective and the robot moves towards it.
For this we have two errors that we need to consider while moving. The first is our actual distance to this point. This distance has a positive or negative signal depending on which side of the trajectory it founds itself in. And the second its a difference the between two angles, where the robot is pointing to and where it should be pointing to. This angles also has a signal depending on which side it points to.

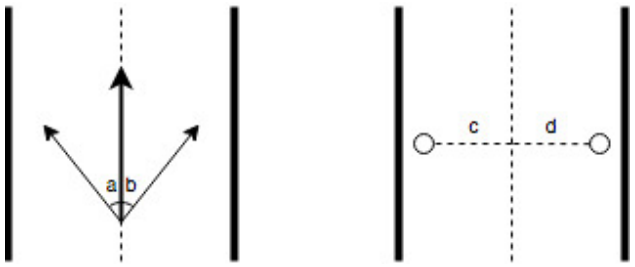


Figure 1: Robot's trajectory error

In Figure 1 on the left the a is the case where the angle has a negative value. And the b where the angle has a positive value.On the right the c is the case where the distance has a negative value. And the d where the distance has a positive value.

Calculating the Movement

After finding the errors the function $CalculateMove(x, y, theta, l, e)$ is called. In it we set the linear velocity to a constant 200(mm/s) and calculate the angular velocity from this formula: $w = k1 * l + k2 * e$.
$k1$ and $k2$ are empirical constants set to 0.6; l is the distance error; and finally, e is the angle error. After this the linear velocity and the angular velocity are applied to the robot. Because the robot doesn't accept angular velocities between 1 and -1, for example 0.1, the velocity is always set to 1 and -1, like so:

$$w = \begin{cases} -1 & \text{if -0.001} > w > \text{ -1} \\ 1 & \text{if 0.001} > w > \text{ 1} \end{cases}$$

This allows us to have more control of the robot to compensate better for small errors. Andso to keep a constant correction of is position. Now we have a acceptable angular velocity to apply to the robot, we use the pioneer_set_controls function with this values.

Reading Sonars

In every interaction the values of the sonars are read, into an array with 8 positions. Each entry corresponds to a given sonar, read from left to right. This means the first entry is the reading of

the sonar closer to the left.This readings have values between 0 and 5000 millimeters, representing the distance of an obstacle in a straight line to the sonar.
This readings can be reflected signals from other sonars, or simply an error. This means that having a reading of 5000 millimeters is considered bad and incorrect.

To avoid this bad readings the used strategy consists of having two global arrays of 8 positions.

The first array is called *PureSensorsArray*, here the average of the last three readings of the sonar is stored, independently of its values. This arrays is useful because it allows to have a notions if the last reading were good or bad.

The second array is called *SensorsArray*, in this array we also do an average of the 3 values, but we only include the readings to the averages if the value of that given sonar is below 4000 millimeters. The only exception are the position 4 and 5, the two front sonars. Their values above 4000mm because they are used to see if the path is clear. In the next section there is the explanation for this two arrays.

Current Position

To be able to follow a trajectory, there is a need to know what is the current position in the map coordinates. The robot gives us the odometry, this is the position it finds itself in since it was turned on. But with time the errors of odometry increases significantly, meaning we need more information to know our position. For this process the aid of the sonars is necessary.
The odometry is consistently reading values, and to those values an offset is added to x, y and *theta*. These offsets are the corrections between what the odometry reads and what the reality is. They are calculated base on the readings of the sonars.

When the robot is making a curve, the sensor readings are turned off. We decided on this because due to the nature of curve, readings from the sensor would be all over the place (robot turning + reflection of the walls). We decided that the odometry within a curve was accurate enough the enter and leave a curve successfully, whereafter the sensors are enabled again.

Odometry

In every interaction the values of the odometry are read using pioneer_read_odometry.
When the robot is turned on the values of odometry are initialized at 0, and when the robot starts to move it changes these values depending on its movements.

When the values are read from odometry an Initial Offset is added. This offset represents the difference between the position of the robot and its actual position in the map. The Initial Offset is a constant and it depends on where we start the robot.

Consequently a second offset *xOdometryOffset*, *yOdometryOffset* and *thetaOdometryOffset* is added, these offset represents the error that the odometry accumulate over time.

Calculating OdometryOffset

To calculate the x and y offset relative to the odometry, it uses the distance from the robot to the left wall. Based on this distance the offset in one of the coordinates is always known with high precision. In figure 3 all the corridors are represented, when the robot is in the corridor 1 and 3 knowing the distance to the wall allows as to calculate the position along the Y-axis, and in corridor 2 and 4 calculate the position along the X-axis. The position is the dimension of the corridor in which the robot is, plus or minus, the distance to the wall. Then the difference between this value and the value that odometry gives is calculated, this becomes the offset for that given axis.

For the offset of the *theta* angle it's assumed that in the middle of an corridor the robot goes straight. The actual value depends on the corridor that it finds itself in. So it's compared to 0, $\frac{-\pi}{2}$, -π, $\frac{-3\pi}{2}$ and -2π in corridor 1, 2, 3 and 4 respectively.

Calculating Left Distance

To obtain the distance from the robot to is left wall, we use the sonars, we try to only use the first sonar (most left one), but because only using 1 sonar might give wrong values we need to adapt, and our strategy is as follows: First we assume that the center of the corridor has a width of 1.67 meters, then we see if the first position of the PureSensorsArray (the left sensor) is below the corridor width, if so, then the left distance is the value in the first position of SensorsArray.

If the value for the first sensor in PureSensorsArray is greater than the corridor width means that is a bad read, the robot is limited to the bounds of the corridor. Then we take a look to the right sensor, the 8th position of PureSensorsArray and do the same check, if the value is acceptable then the left distance is describe in figure 2 where 0.380 is the width of the robot, this is our way to transform the right distance to a left distance.

$$LeftDistance = 1.67 - SensorsArray(8) - 0.380;$$

Figure 2: Left distance, from a right sensor

In the case where the 8th is a bad value, we read the 2th and 7th sensor, usually when 1th and 8th sensor are bad it means the robot is sideways in the corridor, and the 2th or 7th are facing the wall. For this reason we chose the smallest values from the 2th and 7th sensor, and if the smallest value is bellow the corridor width is a god value. In the case we accept the value from 2th sensor this value is the leftdistance, if we accept the 7th then we need to use the equation in figure 2. If all this sensors are bad, we ignore the results of the sensors, and use only odometry.

Front Wall Checking

As a safety and accuracy measure, we added a function that checks if the robots is nearing a corner. When the robot knows that it is close to a corner (from odometry), by its position (x,y) in the map, it will read the front sonar to know how far is from the front wall and correct is position for the curve.

This is done by choosing the minimum value from the position 4 and 5 of SensorsArray, which are the front sensors, if the distance is more than 2 meters, it means we still far away from the corner else that value is the distance to the wall of the corner, and the robot corrects is position.

Emergency stop

For the emergency stop, the robot in every iteration verifies the position 4 and 5 of PureSensorsArray and if any of this values is below 0.3 meters it stops and awaits for an input to continue.

Main Loop

```
for k=1:70000
        [l,e,yy,xx,diff] = getNexPoiny(x,y,theta,xx,yy,diff,150);

        [w,v] = CalculateMove(x,y,theta,l,e);

        MoveRobot(w,v);
        pause(0.1);
        UpdateSonars();
        [x,y,theta] = GetCurrentPosition(xInicial,yInicial,GInicial);

        %finish condition
        if(x <= 2.5 && y <= 25 && y > 23 && AfterHalfHall)
            MoveRobot(0,0);
            break;
        end

end
```

Results

In most of our tests, the robots was successful in traversing the hallways, while it could happen that the robot approached or turned into a wall, it never actually touches them. This happens only sometimes both when leaving the laboratory door and in the second corner. The robots was also successful in avoiding static objects in the hallway (for example, benches). Corners where also successfully implemented due to a well adjusted odometry system in the code.

Furthermore in the last corridor, where $y = 15$, it always makes a sharp turn to the right, but then quickly comes back to its track. This problem is a bug that was not found due to time constraints (we suspect a calculation error when going from calculations with 2π and returning to 0 and it was not necessary to do so because it did not make the robot crash).

Lastly, the biggest problem was returning to the laboratory. Out of the three times during the evaluation the robot was only able to correctly do the turn one time. Nonetheless when helped to do the turn in the right place, as seen in the video sent for evaluation, it was able to then do the turn inside the lab and return to its place and stop on it's own.

Below we can see what a typical test run looks like in function of its position along the world coordinates. And the changes of the angular velocity over time, keeping in mind that the linear velocity is constant.

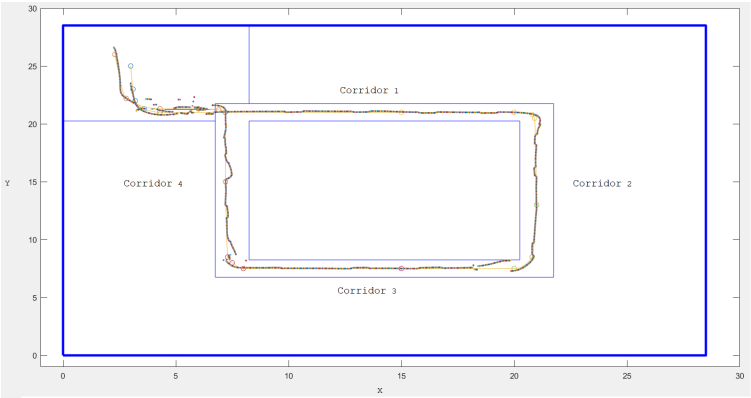


Figure 3: Full lap

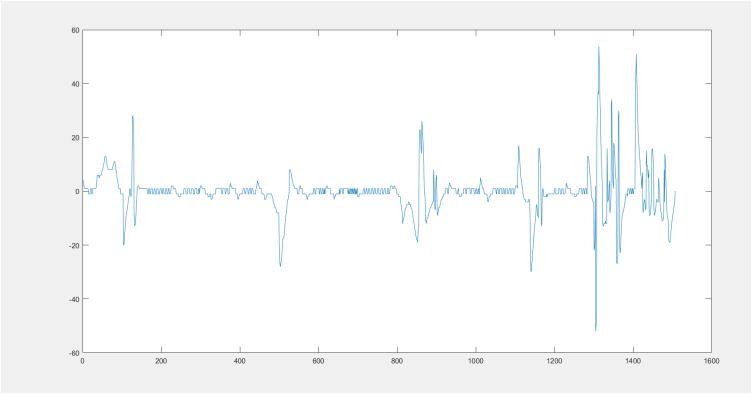


Figure 4: Angular velocity

Conclusion

In the test runs the robot 7 was always used, this is because the code was optimized for it's movements, sensors and errors. To avoid unexpected behaviour, we chose to optimize our code for this robot. As can be seen in the 'Results' section, the robot functioned (almost) fully as intended. While some small bugs and inconsistencies still remain in the code, the majority of its function is

done correctly, we are therefore happy with the final delivered product.

The code could in theory be adapted such that the robot would work in different environments. As long as the obstacles in its path are static (and known beforehand), our current coordinate system could be replaced by coordinates modelling this new environment.